TD #14 — Arbres n-aires, re:ABRs (avec solutions)

Exercice 1. *Fonctions de base sur arbres n-aires*

Dans tout cet exercice, on utilisera le :

```
type 'a ntree = N of 'a * 'a ntree list
```

pour représenter des arbres n-aires en OCaml.

1. Écrivez une fonction OCaml `size : 'a ntree -> int` qui s'évalue en la taille de son argument, définie comme le nombre de nœuds construits avec `N`.

   On propose deux versions :

   ```
   let rec size_forest
       = function
       | [] -> 0
       | x::xs -> (size_tree x) + (size_forest xs)
   and
       size_tree (N (_, kids)) = 1 + (size_forest kids)


   let rec size (N (_, kids)) =
       List.fold_left (+) 1 (List.map size kids)
   ```

2. Écrivez une fonction OCaml `height : 'a ntree -> int` qui s'évalue en la hauteur de son argument, définie récursivement par le fait que la hauteur d'un nœud vaut un de plus que la hauteur maximale de ses enfants (où l'on prendra la convention que le maximum sur l'ensemble vide vaut zéro).

   On propose deux versions :

   ```
   let rec max_height_forest
       = function
       | [] -> 0
       | x::xs -> max (height_tree x) (max_height_forest xs)
   and
       height_tree (N (_, kids)) = 1 + max_height_forest kids

   let rec height (N (_, kids)) =
       1 + (List.fold_left max 0 (List.map height kids))
   ```

3. Écrivez une fonction OCaml :
   `dfs_ante : ('a -> unit) -> 'a ntree -> unit`
   qui applique son premier argument aux étiquettes des nœuds de son second dans un parcours en profondeur préfixe.

   C'est immédiat *via* `List.iter` :

   ```
   let rec dfs_ante f (N (x, kids)) =
       f x ; List.iter (dfs_ante f) kids
   ```

4. Écrivez une fonction OCaml
   `dfs_post : ('a -> unit) -> 'a ntree -> unit`
   qui fait de même pour un parcours en profondeur postfixe.

   Ditto.

   ```
   let rec dfs_post f (N (x, kids)) =
       List.iter (dfs_post f) kids ; f x
   ```

Exercice 2. *Algorithmes d'ABRs à base de* join

Dans tout le début de cet exercice, on utilisera le :

`type 'a tree = N of 'a tree * 'a * 'a tree | E`
pour représenter des arbres binaires en OCaml. Sauf mention du contraire, ces arbres binaires seront supposés être « de recherche », et ne jamais contenir plusieurs nœuds de même étiquette.
Le but de cet exercice est de concevoir des algorithmes pour ABR équilibrés, dont l'opération d'équilibrage est abstraite par une unique fonction *join*. Celle ci satisfait les spécifications suivantes : `join t1 k t2` construit et s'évalue en un arbre dont le parcours en profondeur infixe énumère les valeurs de `t1` dans l'ordre de son parcours en profondeur infixe, puis `k`, puis les valeurs de `t2` dans l'ordre etc. De plus, si `t1` et `t2` sont équilibrés au sens d'un certain schéma d'équilibrage, alors le résultat est équilibré pour le même schéma.

1. En utilisant uniquement la fonction `join` et les manipulations de base du type `'a tree`, écrivez une fonction OCaml :
   `split : 'a -> 'a tree ->  'a tree * bool * 'a tree`
   telle que `split k t` s'évalue en `less, b, more` où `b` vaut `true` ssi. `k` est présent dans `t`, et `less` (resp. `more`) est un ABR dont les éléments sont ceux de `t` strictement inférieurs (resp. supérieurs) à `k`.

   On propose :
   ```
   let rec split k
       = function
       | E -> E, false, E
       | N (_A, b, _C)
           -> if k < b then
                  let _Aless, r, _Amore = split k _A in
                  _Aless, r, (join _Amore b _C) else
              if k > b then
                  let _Cless, r, _Cmore = split k _C in
                  (join _A b _Cless), r, _Cmore else
              _A, true, _C
   ```

2. Toujours avec les mêmes contraintes, écrivez une fonction OCaml
   `splitlast : 'a tree -> 'a tree * 'a`
   telle que pour un argument non vide, `splitlast t` s'évalue en `t', k` où `k` est l'étiquette du nœud « le plus à droite » de `t` (qui est donc de valeur maximale pour les éléments de `t`), et `t'` un arbre d'éléments de mêmes étiquettes que `t`, moins `k`.

   On propose :
   ```
   let rec splitlast
       = function
       | E -> invalid_arg "splitlast: empty tree"
       | N (_A, b, E) -> _A, b
       | N (_A, b, _C) -> let _C', k = splitlast _C in
                          (join _A b _C'), k
   ```

3. Déduisez de ce qui précède une fonction OCaml :
   `join2 : 'a tree -> 'a tree -> 'a tree`
   identique à `join` si ce n'est pour l'absence du paramètre `k:'a`.

   On propose :
   ```
   let join2' t2 = function
       | E -> t2
       | t1 -> let t1', k = splitlast t1 in join t1' k t2

   let join2 t1 t2 = join2' t2 t1
   ```

4. Toujours avec les mêmes contraintes, écrivez une fonction OCaml `add : 'a -> 'a tree -> 'a tree` d'insertion dans un ABR. Le résultat devra être équilibré relativement au schéma implémenté par `join`.

   On a plusieurs possibilités : on peut simplement écrire la fonction d'ajout usuel et utiliser `join` sur les retours d'appels récursifs :

```
let rec add x t
    = match t with
    | E -> node E x E
    | N (_A, b, _C)
        -> if x < b then join (add x _A) b _C else
           if x > b then join _A b (add x _C) else
           t
```

ou utiliser `split` (ce qui est plus court à écrire mais un peu moins efficace) :

```
let add x t =
    let _A, r, _C = split x t in
    if r then t else join _A x _C
```

5. De même pour une fonction OCaml `del : 'a -> 'a tree -> 'a tree` de suppression dans un ABR.

Pareil qu'à la question précédente, deux options :

```
let rec del x t
    = match t with
    | E -> E
    | N (_A, b, _C)
        -> if x < b then join (del x _A) b _C else
           if x > b then join _A b (del x _C) else
           join2 _A _C
```

ou :

```
let del x t =
    let _A, r, _C = split x t in
    if r then join2 _A _C else t
```

On s'intéresse maintenant à une implémentation de `join` pour l'équilibrage AVL. Pour cela, on représente maintenant les arbres binaires par le :

```
type 'a tree = N of int * 'a tree * 'a * 'a tree | E
```

où le constructeur `N` prend un premier argument supplémentaire désignant la hauteur de l'arbre qu'il construit. On supposera également implémenté un constructeur `node : 'a tree -> 'a -> 'a tree -> 'a tree` semi-intelligent qui maintient à jour les informations de hauteur.

6. Écrivez deux fonctions OCaml `rotate_left` et `rotate_right` toutes deux de type `'a tree -> 'a tree` qui effectuent respectivement une rotation à gauche et à droite de leur argument.

On propose :

```
let rotate_left = function
    | N (_, _A, b, (N (_, _C, d, _E))) -> node (node _A b _C) d _E
    | _ -> assert false

let rotate_right = function
    | N (_, (N (_, _A, b, _C)), d, _E) -> node _A b (node _C d _E)
    | _ -> assert false
```

La fonction `join` repose sur deux fonctions symétriques `join_r` et `join_l`, qui effectuent les équilibrages nécessaires quand `t1` ou `t2` est trop haut par rapport à l'autre. On va se concentrer sur l'écriture de `join_r` pour d'équilibrage AVL, qui est de mêmes spécifications que `join` si ce n'est que `t1` et `t2` sont équilibrés AVL t.q. les hauteurs h_1 et h_2 de `t1` et `t2` satisfont $h_2 < h_1 - 1$, et que le résultat est de hauteur $\in [\![h_1, h_1+1]\!]$. Le principe de `join_r` sur des arguments t_1, k, t_2 est le suivant : on déconstruit t_1 (nécessairement non nul) comme $\langle A, b, C\rangle$, et 1) si C est d'une hauteur t.q. $C' := \langle C, k, t_2\rangle$ soit équilibré, on renvoie $\rho\langle A, b, C'\rangle$, où ρ effectue au besoin une (double) rotation quand C' est trop haut par rapport à A (on peut montrer que ce dernier cas se produit uniquement si $h_C = h_A + 1$) ; 2) sinon on appelle `join_r` récursivement sur C, k et t_2 obtenant C', et l'on renvoie $\rho\langle A, b, C'\rangle$, où ρ effectue au besoin une unique rotation.

7. Écrivez une fonction OCaml :
`join_r : 'a tree -> 'a -> 'a tree -> 'a tree`
suivant le principe ci-dessus.

On propose (`join_l` et `join` en bonus) :

```
(* t1, t2 AVL-équlibrés ; h2 < h1 - 1
   hauteur du résultat \in [|h1, h1 + 1|] *)
let rec join_r t1 k t2
    = match t1 with
    | N (_, _A, b, _C)
           (* h1 - 2 <= h_C <= h1 - 1 d'où
              h2 <= h_C  *)
        -> if height _C <= height t2 + 1 then (* h2 <= h_C <= h2 + 1 i.e.
                                                 h_C - 1 <= h2 <= h_C *)
               let _C' = node _C k t2 in (* h_C' = h_C + 1 *)
               if height _A < height _C' - 1 then
               (* h_A = h_C' - 2 *)
               rotate_left @@ node _A b (rotate_right _C') else
               (* h_C' <= h_A <= h_C' + 1 *)
               node _A b _C'
               (* dans les deux cas, la hauteur augmente celle de t1 d'au plus un *)
           else (* h_C > h2 + 1 *)
               let _C' = join_r _C k t2 in (* h_C <= h_C' <= h_C + 1 *)
               let t1k2 = node _A b _C' in
               if height _A < height _C' - 1 then
               rotate_left t1k2 else
               t1k2
    | E -> assert false

(* t1, t2 AVL-équilibrés ; h1 < h2 - 1
   hauteur du résultat \in [|h2, h2 + 1|] *)
let rec join_l t1 k t2
    = match t2 with
    | N (_, _A, b, _C)
           (* h2 - 2 <= h_A <= h2 - 1 d'où
              h1 <= h_A  *)
        -> if height _A <= height t1 + 1 then (* h1 <= h_A <= h1 + 1 i.e.
                                                 h_A - 1 <= h1 <= h_A *)
               let _A' = node t1 k _A in (* h_A' = h_A + 1 *)
               if height _C < height _A' - 1 then
               (* h_C = h_A' - 2 *)
               rotate_right @@ node (rotate_left _A') b _C else
               (* h_A' <= h_C <= h_A' + 1 *)
               node _A' b _C
               (* dans les deux cas, la hauteur augmente celle de t2 d'au plus un *)
           else (* h_A > h2 + 1 *)
               let _A' = join_l t1 k _A in (* h_A <= h_A' <= h_A + 1 *)
               let t1k2 = node _A' b _C in
               if height _C < height _A' - 1 then
               rotate_right t1k2 else
               t1k2
    | E -> assert false

let join t1 k t2 =
    let h1 = height t1 in
    let h2 = height t2 in
    if h2 < h1 - 1 then join_r t1 k t2 else
    if h1 < h2 - 1 then join_l t1 k t2 else
    node t1 k t2
```